System-specific static bug finding: tricks, (bitter) experience, open problems.

Dawson Engler
Stanford

Andy Chou, Ben Chelf, Seth Hallem
Coverity

One-slide of background.

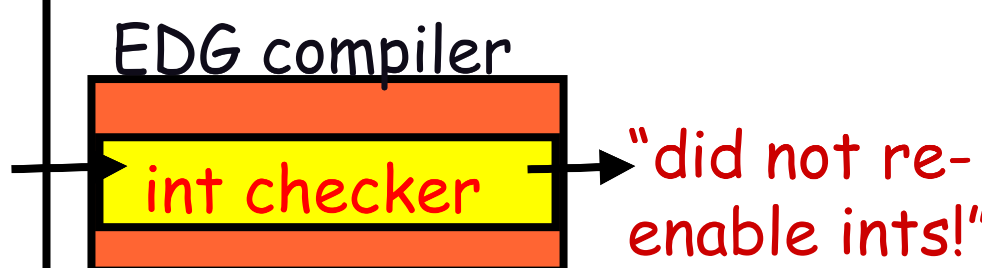
- Academic Lineage
  - MIT: PhD thesis = new operating system (exokernel)
  - Stanford: last seven years developing techniques to find as many serious bugs as possible in large software systems.
  - Co-founded Coverity:100+ customers, cashpositive from T=0
- Our research focuses on three approaches:
  - Implementation-level model checking [OSDI'02,OSDI'04].
  - Automatically generate test cases using symbolic execution [Spin'05, Oakland security'06]
  - System-specific static analysis: use extended compiler to check code. By far the easiest to use and most generally reliable way to find many errors. Rest of the talk on this.

Background: System-specific static analysis

- Systems have many ad hoc correctness rules
  - "acquire lock l before modifying x", "cli() must be paired with sti()," "don't block with interrupts disabled"
  - One error = crashed machine
- If we know rules, can check with extended compiler
  - Rules map to simple source constructs
  - Use compiler extensions to express them

Linux drivers/raid5.c

```
save(flags);
cli();
if(!(buf = kmalloc()))
    return 0;
restore(flags);
return buf;
```

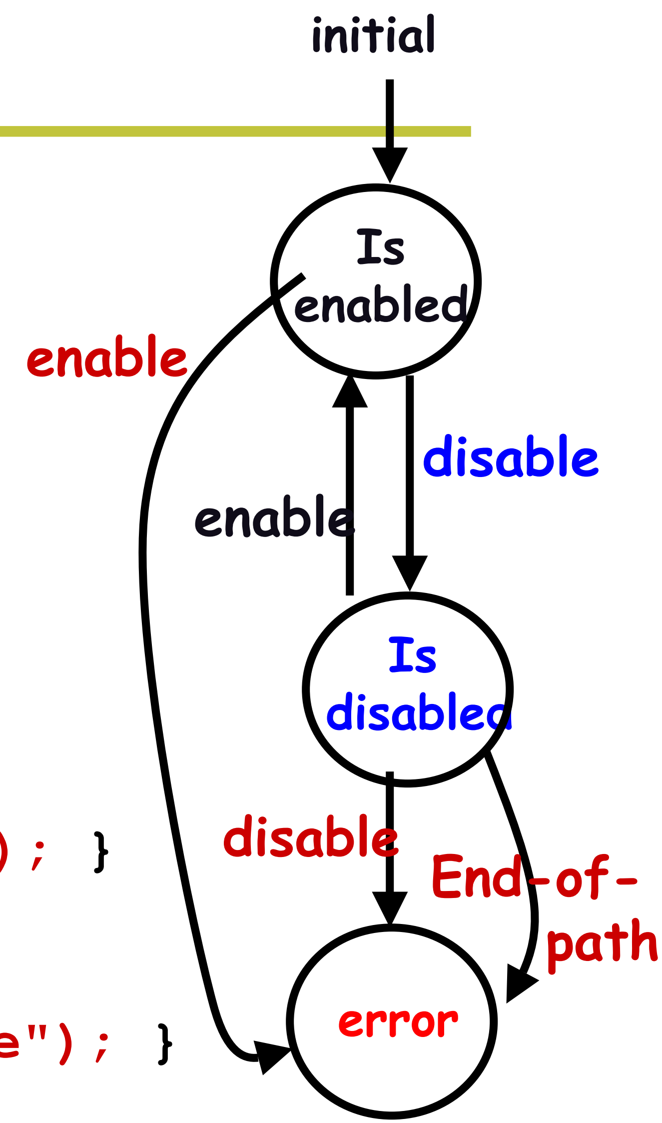


Nice: scales, precise, statically find 1000s of errors

A bit more detail

```
{ #include "linux-includes.h" }
sm chk_interrupts {
 decl { unsigned } flags;
 // named patterns
 pat enable = { sti(); }
        | { restore_flags(flags); };
 pat disable = { cli(); };

 // states
 is_enabled: disable ==> is_disabled
   | enable ==> { err("double enable"); }
   ;
 is_disabled: enable ==> is_enabled
   | disable ==> { err("double disable"); }
   | $end_of_path$ ==>
    { err("exiting w/intr disabled!"); }
   ; }
```

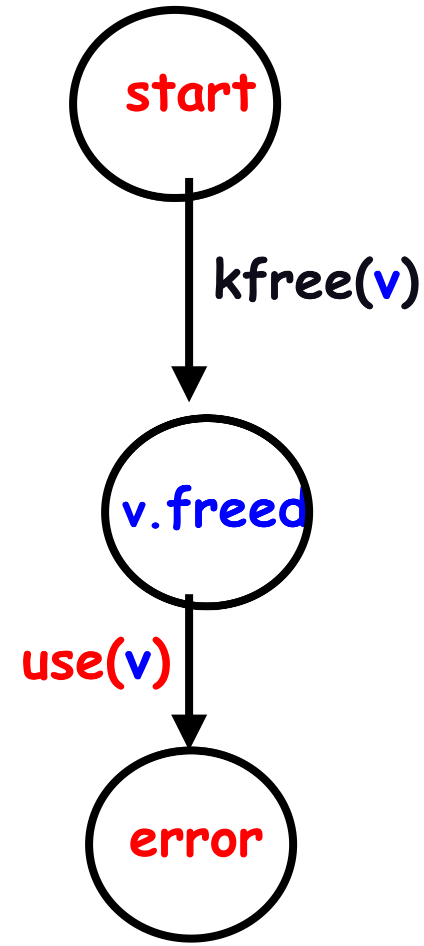


No X after Y: do not use freed memory

```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

 start: { kfree(v); } ==> v.freed
   ;
 v.freed:
    { v != x } || { v == x }
               ==> { /* do nothing */ }
 | { v } ==> { err("Use after free!"); }
    ;
}
```

```
/* 2.4.1: fs/proc/generic.c */
ent->data = kmalloc(…)
if(!ent->data) {
     kfree(ent);
     goto out;
…
out: return ent;
```



High bit: Works well.

- A bunch of checkers:
  - System-specific static checking [OSDI'00] (Best paper)
  - Security checkers [Oakland'02] & annotations [CCS'03]
  - Race conditions and deadlocks [SOSP'03]
  - Path-sensitive memory overflows [FSE'03]
  - Others [ASPLOS'00,PLDI'02,PASTE'02,FSE'02(award)]
  - Statistical: Infer correctness rules [SOSP'01], Z-ranking [SAS'03], Correlation ranking [FSE'03]
- Big system? Always find bugs.
  - New checker, no bugs? Immediate: what's wrong??
- Tenure
- Commercialized(ing): Coverity
  - Successful enough to have a marketing dept.

History of the world as coverity knows it.

Breakthrough technology out of Stanford	Static Source Code Analysis Exercised on Linux	Coverity Incorporated—product further refined on Linux	Company growth and proliferation
1999	**2001**	**2002**	**2003-06**
• Meta-level compilation checker (“Stanford Checker") detects 2000+ bugs in Linux.	• Published several hundred bugs in early version of Linux. • Hundreds of defects fixed by the Linux community	• Deluge of requests from companies wanting access to the new technology. • Linux work continues: More than 2000 bugs found • Created Linuxbugs site as a free service	• 100 customers including Juniper, Synopsys, Oracle, Veritas, nVidia, palmOne. • IDC: Coverity is the fastest growing software quality tools vendor—and in the top 10.

A partial list of 100 customers…

EDA
- Synopsys
- Mentor Graphics
- cadence

A&D
- NASA — National Aeronautics and Space Administration
- Jet Propulsion Laboratory, California Institute of Technology

Automotive
- DENSO
- Xanavi

Networking
- Cisco Systems
- hp invent
- Juniper Networks
- Marconi
- ciena
- Avici Systems

Storage
- Veritas
- EMC² where information lives
- panasas

Telco
- AudioCodes
- tellabs
- france telecom
- Excel — Where Applications Perform Best

Security
- McAfee
- PGP
- Check Point Software Technologies Ltd.

OS
- Wind River
- symbian

Wireless
- Kyocera
- Nokia — Connecting People
- Ericsson
- palm

coverity

Prevent Library of Checkers

Quality
- System and Process Crash
- Memory/Resource Leaks
- Data, Memory, File Corruption
- Performance Degradation
- Unpredictable Behavior

Concurrency
- Deadlocks
- Lock Contention
- Unpredictable performance
- Performance degradation

Security
- Denial of Service
- Privilege Escalation
- Malicious Code

Checker Library

Resource Problems
- Resource Leak
  - Memory leak
  - File pointer leak
  - System resource leak

Concurrency Problems
- Double Lock
- Missing unlock
- Incorrect lock acquisition
- Sleeping while locked

API Usage Errors
- Passing large parameters
- Insecure temp file creation
- Improper method override

Pointer Errors
- Use of uninitialized data
  - Uninitialized memory
  - Uninitialized variable
- Mismatched allocation operators
- Dereferencing invalid pointers
  - Null pointer dereference
  - Wrong address space
  - Accessing freed pointers
- Dangling stack references
- Use of freed resource
  - Double free (memory, file pointers, system resources)
  - Use after free (memory, file pointers, system resources)

Logic Errors
- Flawed branch logic
- Use of invalid STL iterators
- Useless operation
- Inconsistent error handling
- Security logic errors
  - Time of check, time of use
  - Insecure file creation
  - Improper chroot
  - Improper privilege inheritance

Security Warnings
- Potentially insecure coding practices

Bounds Errors
- Out of bounds array access
- Buffer underflow
- Stack smashing
  - Stack overflow
  - Stack buffer overrun
  - Stack string overrun
- Bad negative integer cast
- Incorrect allocation size
- Non-null terminated strings

External Data Handling
- Integers
  - Loop bound
  - Array access
  - Allocation size
- Strings
  - Buffer overflow
  - SQL Injection
  - Format string errors
  - Cross-site scripting

Talk overview

- System-specific static analysis
  - Correctness rules map clearly to concrete source actions
  - Check by making compilers aggressively system-specific
  - Nice: One person writes checker, imposed on all code.
- Next: Belief analysis
  - Using programmer beliefs to infer state of system and rules to check
  - Key: Find bugs without knowing truth.
- General experiences + open problems.
- Weird things that happen when academics try to commercialize a static checking tool.

Goal: find as many serious bugs as possible

- Problem: what are the rules?!?!
  - 100-1000s of rules in 100-1000s of subsystems.
  - To check, must answer: Must a() follow b()? Can foo() fail? Does bar(p) free p? Does lock l protect x?
  - Manually finding rules is hard. So don't. Instead infer what code believes, cross check for contradiction
- Intuition: how to find errors without knowing truth?
  - Contradiction. To find lies: cross-examine. Any contradiction is an error.
  - Deviance. To infer correct behavior: if 1 person does X, might be right or a coincidence. If 1000s do X and 1 does Y, probably an error.
  - Crucial: we know contradiction is an error without knowing the correct belief!

Cross-checking program belief systems

- MUST beliefs:

  Inferred from acts that imply beliefs code *must* have.

```
x = *p / z; // MUST belief: p not null
            // MUST: z != 0
unlock(l);  // MUST: l acquired
x++;        // MUST: x not protected by l
```

  Check using internal consistency: infer beliefs at different locations, then cross-check for contradiction

- MAY beliefs: could be coincidental

  Inferred from acts that imply beliefs code *may* have

```
A();  A();  A();  A();
                        // MAY: A() and B()
...   ...   ...   ...
B();  B();  B();  B();  // must be paired
```

```
B(); // MUST: B() need not
     // be preceded by A()
```

  Check as MUST beliefs; rank errors by belief confidence.

Internal Consistency: finding security holes

- Applications are bad:
  - Rule: "do not dereference user pointer <p>"
  - One violation = security hole
  - Detect with static analysis if we knew which were "bad"
  - Big Problem: which are the user pointers???
- Sol'n: forall pointers, cross-check two OS beliefs
  - "*p" implies safe kernel pointer
  - "copyin(p)/copyout(p)" implies dangerous user pointer
  - Error: pointer p has both beliefs.
  - Implemented as a two pass global checker
- Result: 24 security bugs in Linux, 18 in OpenBSD
  - (about 1 bug to 1 false positive)

An example

- Still alive in linux 2.4.4:

```
/* drivers/net/appletalk/ipddp.c:ipddp_ioctl */
case SIOCADDIPDDPRT:
      return ipddp_create(rt);
case SIOCDELIPDDPRT:
      return ipddp_delete(rt);
case SIOFCINDIPDDPRT:
      if(copy_to_user(rt, ipddp_find_route(rt)
                      sizeof(struct ipddp_route)))
            return -EFAULT;
```

Tainting marks "rt" as a tainted pointer, checker warns that rt is passed to a routine that dereferences it

2 other examples in same routine…

MAY beliefs

- Separate fact from coincidence? General approach:

  Assume MAY beliefs are MUST beliefs.

  Check them

  Count number of times belief passed check (S=success)

  Count number of times belief failed check (F=fail)

  Expect: valid beliefs = high ratio of S to F.

  Use S and F to compute confidence that belief is valid.

  Rank errors based on this confidence.

  Go down list, inspecting until false positives are too high.

- How to weigh evidence?

How to weigh MAY beliefs

- Wrong way: percentage. (Ignores population size)
  - Success=1, Failure=0, Percentage = 1/1 * 100= 100%
  - Success=990, Failure=10, Percentage = 990/1000 = 99%
- A better way: "hypothesis testing."
  - Treat each check as independent binary coin toss
  - Pick probability p0 that coin "coincidently" comes up S.
  - For a given belief, compute how "unlikely" that it coincidently got S successes out of N (N=S+F) attempts

$$Z = (\text{observed} - \text{expected}) / \text{stderr}$$
$$= (S - N*p0) / \text{sqrt}(N*p0*(1-p0))$$

- HUGE mistake: pick T, where Z>T implies MUST
  - Becomes very sensitive to T.

Statistical: Deriving deallocation routines

- Use-after free errors are horrible.
  - Problem: lots of undocumented sub-system free functions
  - Soln: derive behaviorally: pointer "p" not used after call "foo(p)" implies MAY belief that "foo" is a free function
- Conceptually: Assume all functions free all arguments
  - (in reality: filter functions that have suggestive names)
  - Emit a "check" message at every call site.
  - Emit an "error" message at every use

foo(p);	foo(p);	foo(p);	bar(p);	bar(p);	bar(p);
*p = x;	*p = x;	*p = x;	p = 0;	p = 0;	*p = x;

  - Rank errors using z test statistic: z(checks, errors)
  - E.g., foo.z(3, 3) < bar.z(3, 1) so rank bar's error first
  - Results: 23 free errors, 11 false positives

Talk Overview

- Belief analysis: broader checking
  - **Beliefs code MUST have: Contradictions = errors**
  - **Beliefs code MAY have: check as MUST beliefs and rank errors by belief confidence**
  - **Key feature: find errors without knowing truth**
- Rest of talk:
  - **Weird things that happen when academics try to commercialize static checking.**
  - **General experience.**

Weird things that surprise academics trying to commercialize a static checking tool.

Andy Chou, Ben Chelf, Seth Hallem
Charles Henri-Gros, Bryan Fulton, Ted Unangst
Chris Zak
Coverity

Dawson Engler
Stanford

A naïve view

- Initial market analysis:
  - "We handle Linux, BSD, we just need a pretty box!"
  - Not quite.
- First rule of static analysis: no check, no bug.
  - Two first order examples we never would have guessed.
  - Problem 1: if you can't find the code, can't check it.
  - Problem 2: if you can't compile code, you can't check it.
- And then: how to make money on software tool?
  - "Tools. Huh. Tools are hard." Any VC in early 2000.

Myth: the C (or C++) language exists.

- Well, not really. The standard is not a compiler.
  - What exists: gcc-2.1.9-ac7-prepatch-alpha, xcc-i-did-not-understand-pages4,33,208-242-of-standard.
  - Oh. And Microsoft. Conformance = competitive disadvantage. Do the math on how this deforms .c files
  - Basic LALR law: What can be parsed will be written.
- Rule: static analysis must compile code to check.
  - If you cannot (correctly) parse "language" cannot check.

  - Common (mis)usage model: "allegedly C" header file does something bizarre not-C thing. Included by all source. Customer watches your compiler emit voluminous parse errors. (This is not impressive.)
  - Of course: gets way worse with C++ (which we support)

Some bad examples to find in headers

- Banal. But take more time than you can believe:

```
void x;
```

```
short x; int *y = &(int)x;
```

```
int foo(int a, int a);
```

```
unsigned x @ "TEXT";
```

```
unsigned x = 0xdead_beef;
```

```
Int16 ErrSetJump(ErrJumpBuf buf) = { 0x4E40 + 15, 0xA085 };
```

- And, of course, asm:

```
#pragma asm
    mov eax, eab
#pragma end_asm
```

```
asm foo() {
        mov eax, eab;
}
```

```
// newline = end
__asm mov eax, eab

// "]" = end
__asm [
        mov eax, eab
    ]
```

Microsoft example: precompiled headers

- Spec:

> The compiler treats all code occurring before the .h file as precompiled. It skips to just beyond the #include directive associated with the .h file, uses the code contained in the .pch file, and then compiles all code after filename

- Implication

```
I can put whatever I want here.
It doesn't have to compile.
If your compiler gives an error it sucks.
#include <some-precompiled-header.h>
```

- It gets worse: on-the-fly header fabrication

Solution: pre-preprocessing rewrite rules.

- Supply regular expressions to rewrite bad constructs

```
#pragma asm
...
#pragma end_asm
```

↓

```
ppp_translate ("/#pragma asm/#if 0/");
ppp_translate("/#pragma end_asm/#endif/");
```

↓

```
#if 0
...
#endif
```

What this all means concretely.

- We use Edison Design Group (EDG) frontend
  - Pretty much everyone uses. Been around since 1989.
  - Aggressive support for gcc, microsoft, etc. (bug compat!)
- Still: coverity by far the largest source of EDG bugs:
  - 146 parsing test cases (i.e., we got burned)
  - 219 compiler line translation test cases (i.e., ibid).
  - 163 places where frontend hacked ("#ifdef COVERITY")
  - Still need custom rewriter for many supported compilers:

205 hpux_compilers.c
215 iar_compiler.c
240 ti_compiler.c
251 green_hills_compiler.c
377 intel_compilers.c
453 diab_compilers.c

453 sun_compilers.c
485 arm_compilers.c
617 gnu_compilers.c
748 microsoft_compilers.c
1587 metrowerks_compilers.c
…

Academics don't understand money.

- "We'll just charge per seat like everyone else"
  - **Finish the story: "Company X buys three Purify seats, one for Asian, one for Europe and one for the US…"**
- Try #2: "we'll charge per lines of code"
  - **"That is a really stupid idea: (1) …, (2) … , … (n) …"**
  - **Actually works. I'm still in shock. Would recommend it.**
- Good feature for seller:
  - **No seat games. Revenue grows with code size. Run on another code base = new sale.**
- Good feature for buyer: No seat-model problems
  - **Buy once for project, then done. No per-seat or per-usage cost; no node lock problems; no problems adding, removing or renaming developers (or machines)**
  - **People actually seem to like this pitch.**

Some experience.

- Surprise: Sales guys are great
  - **Easy to evaluate. Modular.**
- Company X buys tool, then downsizes.
  - **Good or bad?**
- Large companies "want" to be honest
  - **Veritas: want monitoring so don't accidently violate!**
- What can you sell?
  - **User not same as tool builder. Naïve. Inattentive. Cruel.**
  - **Makes it difficult to deploy anything sophisticated.**
  - **Example: statistical inference, race conditions.**
  - **Some ways, checkers lag much behind our research ones.**

"No, your tool is broken: that's not a bug"

- "No, the loop will go through once!"

```
for(i=1; i < 0; i++) {
   ...deadcode...
}
```

```
for(s=0; s < n; s++) {
   ...
   switch(s) {
     case 0: assert(0);
             return;
     ...
   }
   ...dead code...
```

- "No, && is 'or'!"

```
void *foo(void *p, void *q) {
    if(!p && !q)
        return 0;
```

- "No, ANSI lets you write 1 past end of the array!"

  ("We'll have to agree to disagree."   !!!!)

```
unsigned p[4];    p[4] = 1;
```

Laws of static bug finding

- Vacuous tautologies that imply trouble
  - **Can't find code, can't check.**
  - **Can't compiler code, can't check.**
- A nice, balancing empirical tautology
  - **If can find code**
  - **AND checked system is big**
  - **AND can compile (enough) of it**
  - **THEN: will always find serious errors.**
- A nice special case:
  - **Check rule never checked? Always find bugs. Otherwise immediate kneejerk: what wrong with checker???**

Some cursory static analysis experiences

- Bugs are everywhere
  - **Initially worried we'd resort to historical data…**
  - **100 checks? You'll find bugs (if not, bug in analysis)**
- Finding errors often easy, saying why is hard
  - **Have to track and articulate all reasons.**
- Ease-of-inspection *crucial*
  - **Extreme: Don't report errors that are too hard.**
- The advantage of checking human-level operations
  - **Easy for people? Easy for analysis. Hard for analysis? Hard for people.**
- Soundness not needed for good results.

Myth: more analysis is always better

- Does not always improve results, and can make worse
- The best error:
  - Easy to diagnose
  - True error
- More analysis used, the worse it is for both
  - More analysis = the harder error is to reason about, since user has to manually emulate each analysis step.
  - Number of steps increase, so does the chance that one went wrong. No analysis = no mistake.
- In practice:
  - Demote errors based on how much analysis required
  - Revert to weaker analysis to cherry pick easy bugs
  - Give up on error classes that are too hard to diagnose.

No bug is too stupid to check for.

- Someone, somewhere will do anything you can think of.
- Best recent example:

  From security patch for bug found by Coverity in X windows that lets almost any local user get root.

```
--- hw/xfree86/common/xf86Init.c.orig 2006-03-17...
   /* First the options that are only allowed for root */
-     if (getuid() != 0 && geteuid == 0) {
+     if (getuid() != 0 && geteuid() == 0) {
   ErrorF("-configure can only be used by root.\n");
   exit(1);
    }
```

- Next: Two amazingly effective checks.

One of the best stupid checks: Deadcode

- Programmer generally intends to do useful work.

  Use constraint analysis to flag code where all paths to it are impossible. Often serious logic bug.

- From UU aodv (good code):

  Linked list removal mistake. After send, take packet off queue. Bug = if any packets on list before the one we want will lose them!

```
// packet_queue.c:packet_queue_send
prev = null;
while(curr) {
    if(curr->dst_addr == dst_addr) {
        if(prev == NULL)
            PQ.head = curr->next;
        else
         ...DEADCODE [prev never updated]...
```

Internal null: trivial, amazingly effective.

- "*p" implies programmer believes p is not null
- A check (p == NULL) implies two beliefs:
  - **POST: p is null on true path, not null on false path**
  - **PRE: p was unknown before check**
- Cross-check beliefs: contradiction = error.

- Check-then-use (79 errors, 26 false pos)

```
/* 2.4.1: drivers/isdn/svmb1/capidrv.c */
if(!card)
  printk(KERN_ERR, "capidrv-%d: …", card->contrnr…)
```

Null pointer fun

- Use-then-check: 102 bugs, 4 false

```
/* 2.4.7: drivers/char/mxser.c */
struct mxser_struct *info = tty->driver_data;
unsigned flags;
if(!tty || !info->xmit_buf)
      return 0;
```

- Contradiction/redundant checks (24 bugs, 10 false)

```
/* 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt)
   return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);
/* REDUNDANT check */
if(!fb_info.regbase_virt) {
   iounmap(fb_info.regbase_virt);
```

Assertion: Soundness is often a distraction

- Soundness: Find all bugs of type X.
  - Not a bad thing. More bugs good.
  - BUT: can only do if you check weak properties.
- What soundness really wants to be when it grows up:
  - Total correctness: Find all bugs.
  - Most direct approximation: find as many bugs as possible.
- Opportunity cost:
  - Diminishing returns: Initial analysis finds most bugs
  - Spend time on what gets the next biggest set of bugs
  - Easy experiment: bug counts for sound vs unsound tools.
- Soundness violates end-to-end argument:
  - "It generally does not make much sense to reduce the residual error rate of one system component (property) much below that of the others."

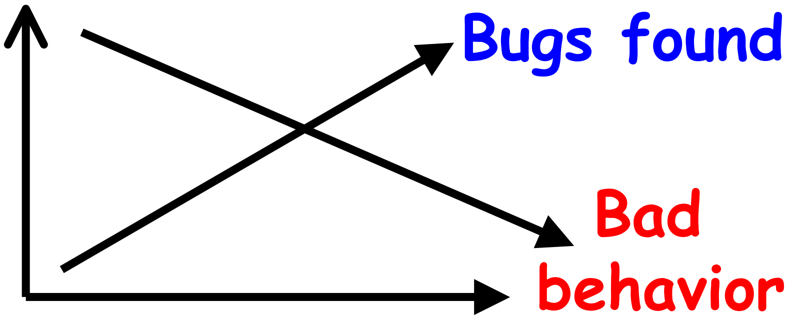
Static vs dynamic bug finding

- Static: precondition = compile (some) code.
  - All paths + don't need to run + easy diagnosis.
  - Low incremental cost per line of code
  - Can get results in an afternoon.
  - 10-100x more bugs.
- Dynamic: precondition = compile all code + run
  - What does code do? How to build? How to run?
  - Runs code, so can check implications.
  - Good: Static detects ways to cause error, dynamic can check for the error itself.
- Result:
  - Static better at checking properties visible in source, dynamic better at properties implied by source.

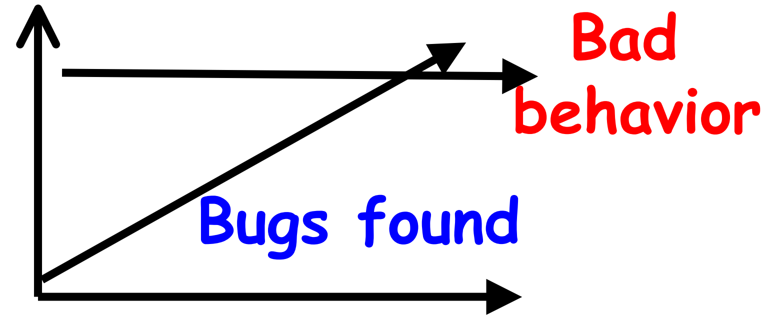
Open Q: how to get the bugs that matter?

- Myth: all bugs matter and all will be fixed
  - *FALSE*
  - Find 10 bugs, all get fixed. Find 10,000…
- Reality
  - All sites have many open bugs (observed by us & PREfix)
  - Myth lives because state-of-art is so bad at bug finding
  - What users really want: The 5-10 that "really matter"
- General belief: bugs follow 90/10 distribution
  - Out of 1000, 100 (10? or 1?) account for most pain.
  - Fixing 900+ waste of resources & may make things worse
- How to find worst? No one has a good answer to this.
  - Possibilities: promote bugs on executed paths or in code people care about, …

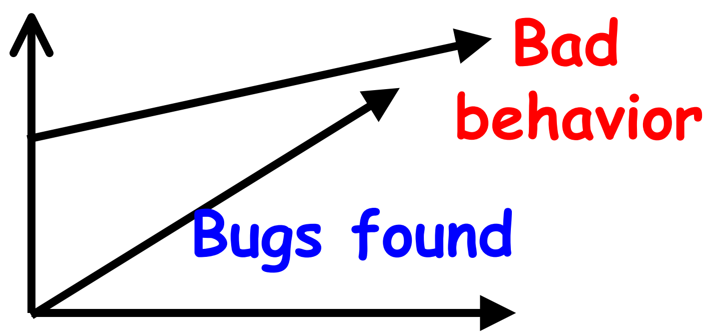
Open Q: Do static tools really help?



The optimistic hope



The null hypothesis



An Ugly Possibility

Danger: Opportunity cost.

Danger: Deterministic canary bugs to non-deterministic.

Summary

- Effective static analysis of real code
  - **Write small extension, apply to code, find 100s-1000s of bugs in real systems**
  - **Result: Static, precise, immediate error diagnosis**
  - **One person writes, imposes on all code.**
- Belief analysis: broader checking
  - **Using programmer beliefs to infer state of system, relevant rules**
  - **Key feature: find errors without knowing truth**
- Found lots of serious bugs everywhere.

- Free trial (or job!):
  - **www.coverity.com**